A Bisimulation for Dynamic Sealing

Eijiro Sumii
University of Pennsylvania
sumii@saul.cis.upenn.edu

Benjamin C. Pierce
University of Pennsylvania
bcpierce@cis.upenn.edu

Abstract

We define $\lambda_{\texttt{seal}}$, an untyped call-by-value λ-calculus with primitives for protecting abstract data by *sealing*, and develop a bisimulation proof method that is sound and complete with respect to contextual equivalence. This provides a formal basis for reasoning about data abstraction in open, dynamic settings where static techniques such as type abstraction and logical relations are not applicable.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*abstract data types*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; F.3 [**Theory of Computation**]: Logics and Meanings of Programs; E.3 [**Data**]: Data Encryption; C.2.2 [**Computer-Communication Networks**]: Network Protocols—*protocol verification*

General Terms

Theory, Languages, Security

1 Introduction

Dynamic sealing: Birth, death, and rebirth

Sealing is a linguistic mechanism for protecting abstract data. As originally proposed by Morris [18, 19], it consists of three constructs: seal creation, sealing, and unsealing. A fresh seal is created for each module that defines abstract data. Data is sealed when it is passed out of the module, so that it cannot be inspected or modified by outsiders who do not know the seal; the data is unsealed again when it comes back into the module, so that it can be manipulated concretely. Data abstraction is preserved as long as the seal is kept local to the module.

POPL'04, January 14–16, 2004, Venice, Italy.

Originally, sealing was a dynamic mechanism. Morris also proposed a static variant [19], in which the creation and use of seals at module boundaries follow a restricted pattern that can be verified by the compiler, removing the need for run-time sealing and unsealing. Other researchers found that a similar effect could be obtained by enriching a static type system with mechanisms for *type abstraction* (see CLU [13], for example). Type abstraction became the primary method for achieving data abstraction in languages from CLU to the present day. It is also well understood via the theory of existential types [16].

Recently, however, as programming languages and the environments in which they operate become more and more open—e.g., addressing issues of persistence and distribution—dynamic sealing is being rediscovered. For example, Rossberg [26] proposes to use a form of dynamic sealing to allow type abstraction to coexist with dynamic typing; Leifer et al. [12] use hashes of implementations of abstract types to protect abstractions among different programs running on different sites; Dreyer et al. [7] use a variant of sealing (somewhere between static and dynamic) to give a type-theoretic account of ML-like modules and functors; finally, we [21] have proposed a translation (conjectured to be fully abstract) of System-F-style type abstraction into dynamic sealing.

Another reason for the renewal of interest in sealing is that it happens to coincide with *perfect encryption* (under shared-key cryptography), that is, with an ideal encryption scheme where a ciphertext can be decrypted only if the key under which it was encrypted is known explicitly. Perfect encryption is a common abstraction in current research on both systems security and programming languages, for example in modeling and reasoning about cryptographic protocols (e.g., the spi-calculus [3]).

Problem

Although interest in dynamic sealing is reviving, there remains a significant obstacle to its extensive study: the lack of sufficiently powerful methods for reasoning about sealing. First, to the best of our knowledge, there has been no work at all on proof techniques for sealing in untyped sequential languages. (There are several versions of bisimulation for the spi-calculus, but encoding other languages such as λ-calculus into spi-calculus raises the question of what abstraction properties are preserved by the encoding itself.) Second, even in statically typed settings, the published techniques for obtaining abstraction properties are in general very weak. For instance, the first two [12, 26] of the works cited above use (variants of) the colored brackets of Zdancewic et al. [8, 32] but only prove (or even state) abstraction properties for cases where abstract data is published *by itself* with no interface functions provided (i.e., once sealed, data is never unsealed).

Abstraction as equivalence

We aim to establish a method for proving abstraction of programs using dynamic sealing in an untyped setting. To this end, let us first consider how to state the property of abstraction in the first place. Take, for example, the following module implementing complex numbers in an imaginary ML-like language.

```
module PolarComplex =
  abstype t = real * real
  let from_re_and_im : real * real -> t =
    fun (x, y) ->
      (sqrt(x * x + y * y), atan2(y, x))
  let to_re_and_im : t -> real * real =
    fun (r, t) ->
      (r * cos(t), r * sin(t))
  let multiply : t * t -> t =
    fun ((r1, t1), (r2, t2)) ->
      (r1 * r2, t1 + t2)
end
```

Using dynamic sealing instead of type abstraction, this module can be written as follows for some secret seal `k`.

```
module PolarComplex =
  let from_re_and_im =
    fun (x, y) ->
      let z = (sqrt(x * x + y * y), atan2(y, x)) in
      <seal z under k>
  let to_re_and_im =
    fun z ->
      let (r, t) = <unseal z under k> in
      (r * cos(t), r * sin(t))
  let multiply =
    fun (z1, z2) ->
      let (r1, t1) = <unseal z1 under k> in
      let (r2, t2) = <unseal z2 under k> in
      let z = (r1 * r2, t1 + t2) in
      <seal z under k>
end
```

Now, the question is: Is this use of sealing correct? That is, does it really protect data abstraction? In particular, can we show that this module has the same external behavior as another sealed module that also implements complex numbers, e.g., the following module with another secret seal `k'`?

```
module CartesianComplex =
  let from_re_and_im =
    fun (x, y) ->
      <check that x and y are real numbers>;
      let z = (x, y) in <seal z under k'>
  let to_re_and_im =
    fun z ->
      let (x, y) = <unseal z under k'> in (x, y)
  let multiply =
    fun (z1, z2) ->
      let (x1, y1) = <unseal z1 under k'> in
      let (x2, y2) = <unseal z2 under k'> in
      let z = (x1 * x2 - y1 * y2,
               x1 * y2 + x2 * y2) in
      <seal z under k'>
end
```

Formally, we want to show the contextual equivalence [17] of the two modules `PolarComplex` and `CartesianComplex`. In general, however, it is difficult to *directly* prove contextual equivalence because it demands that we consider an infinite number of contexts.

Equivalence by bisimulation

To overcome this difficulty, we define a notion of *bisimulation* for our language (by extending applicative bisimulation [4]) and use it as a tool for proving contextual equivalence. Essentially, a bisimulation records a set of pairs of "corresponding values" of two different programs. In the example of `PolarComplex` and `CartesianComplex`, the bisimulation is (roughly):

$$
\begin{array}{l}
\{(\texttt{PolarComplex}, \texttt{CartesianComplex})\} \cup \\
\{(\texttt{PolarComplex.from\_re\_and\_im}, \\
\quad \texttt{CartesianComplex.from\_re\_and\_im}), \\
\quad (\texttt{PolarComplex.to\_re\_and\_im}, \\
\quad \texttt{CartesianComplex.to\_re\_and\_im}), \\
\quad (\texttt{PolarComplex.multiply}, \\
\quad \texttt{CartesianComplex.multiply})\} \cup \\
\{((x,y),(x,y)) \mid x,y \text{ real numbers}\} \cup \\
\{(\{(r,\theta)\}_k, \{(r\cos\theta, r\sin\theta)\}_{k'}) \mid r \geq 0\}
\end{array}
$$

The first part is the modules themselves. The second part is the individual elements of the modules. The third is arguments of `from_re_and_im` as well as results of `to_re_and_im`. The last is the representations of complex numbers sealed under k or k', where $\{\ \}$ denotes sealing.

From the soundness of bisimulation, we obtain the contextual equivalence of the two modules. Furthermore, our bisimulation is *complete*: if two programs are contextually equivalent, then there always exists a bisimulation between them. This means that (at least in theory) we can use bisimulation to prove *any* valid contextual equivalence.

Contribution

The main contribution of this work is a sound and complete bisimulation proof method for contextual equivalence in an untyped functional language with dynamic sealing. Along the way, we are led to refine the usual contextual equivalence to account for the variations in observing power induced by the context's knowledge (or ignorance) of the seals used in observed terms.

Parts of our theory are analogous to bisimulation techniques developed for the spi-calculus [1, 2, 5, 6]. However, our bisimulation is technically simpler and thus more suitable for reasoning about dynamic sealing for data abstraction in sequential languages. Furthermore, our setting requires us to extend even the definition of contextual equivalence in a natural but significant way, as discussed in Section 3.

Structure of the paper

The rest of this paper is structured as follows. Section 2 formalizes the syntax and the semantics of our language, $\lambda_{\texttt{seal}}$. Section 3 defines a suitable notion of contextual equivalence. Section 4 presents our bisimulation and gives several examples, including the complex number packages discussed above and an encoding of the Needham-Schroeder-Lowe key exchange protocol. Section 5 proves soundness and completeness of the bisimulation with respect to contextual equivalence. Section 7 discusses related work, and Section 8 sketches future work.

$d, e ::=$	term
x	variable
$\lambda x.e$	function
$e_1\, e_2$	application
c	constant
$op(e_1, \ldots, e_n)$	primitive
$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$	conditional branch
$(e_1, \ldots, e_n)$	tupling
$\#_i(e)$	projection
k	seal
$\nu x.e$	fresh seal generation
$\{e_1\}_{e_2}$	sealing
$\texttt{let}\ \{x\}_{e_1} = e_2\ \texttt{in}\ e_3\ \texttt{else}\ e_4$	unsealing
$u, v, w ::=$	value
$\lambda x.e$	function
c	constant
$(v_1, \ldots, v_n)$	tuple
k	seal
$\{v\}_k$	sealed value

Figure 1. Syntax of $\lambda_{\texttt{seal}}$

Notation

Throughout the paper, we use overbars as shorthands for sequences—e.g., we write $\overline{x}$ and $(\overline{v}, \overline{v'})$ instead of $x_1, \ldots, x_n$ and $(v_1, v_1'), \ldots, (v_n, v_n')$ where $n \geq 0$. Similarly, $\{\overline{k}\}$ is a shorthand for the set $\{k_1, \ldots, k_n\}$ where $k_i \neq k_j$ for any i and j. When s and t are sets, $s \uplus t$ is defined to be $s \cup t$ if $s \cap t = \emptyset$, and undefined otherwise.

2 Syntax and Semantics

$\lambda_{\texttt{seal}}$ is the standard untyped, call-by-value λ-calculus extended with sealing. Its syntax is given in Figure 1. Seal k is an element of the countably infinite set K of all seals. We use meta-variables s and t for finite subsets of K. Fresh seal generation $\nu x.e$ generates a fresh seal k, binds it to x and evaluates e. The meaning of freshness will soon be clarified below. Sealing $\{e_1\}_{e_2}$ evaluates e_1 to value v and e_2 to seal k, and seals v under k. Unsealing $\texttt{let}\ \{x\}_{e_1} = e_2\ \texttt{in}\ e_3\ \texttt{else}\ e_4$ evaluates e_1 to seal k_1 and e_2 to sealed value $\{v\}_{k_2}$. If $k_1 = k_2$, the unsealing succeeds and e_3 is evaluated with x bound to v. Otherwise, the unsealing fails and e_4 is evaluated.

The calculus is also parametrized by first-order constants and primitives such as real numbers and their arithmetics. We use infix notations for binary primitives like $e_1 + e_2$. We assume that constants include booleans `true` and `false`. We also assume that *op* includes the equality $=$ for constants.

We adopt the standard notion of variable binding and write $FV(e)$ for the set of free variables in e. We also write $Seals(e)$ for the set of seals that appear in term e.

We write $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ for $(\lambda x.e_2)e_1$. We also write $\bot$ for $(\lambda x.xx)(\lambda x.xx)$ and $\lambda\{x\}_k.e$ for $\lambda y.\texttt{let}\ \{x\}_k = y\ \texttt{in}\ e\ \texttt{else}\ \bot$ where $y \notin FV(e)$. Furthermore, we write $\lambda(x,y).e$ for $\lambda z.\texttt{let}\ x = \#_1(z)\ \texttt{in let}\ y = \#_2(z)\ \texttt{in}\ e$ where $z \notin FV(e)$. We use similar notations of pattern matching throughout the paper.

The semantics of $\lambda_{\texttt{seal}}$ is given in Figure 2 by big-step evaluation $\langle s \rangle\, e \Downarrow \langle t \rangle\, v$ annotated with the set s of seals before the evaluation and the set t of seals after the evaluation. It is parametrized by the meaning $[\![op(c_1, \ldots, c_n)]\!]$ of primitives. For example, $[\![1.23 + 4.56]\!] = 5.79$. For simplicity, we adopt the left-to-right evaluation order. As usual, substitutions $[e/x]$ avoid capturing free variables by implicit α-conversion. The meaning of freshness is formalized by requiring $k \notin s$ in (E-New). We write $\langle s \rangle\, e \Downarrow$ if $\langle s \rangle\, e \Downarrow \langle t \rangle\, v$ for some t and v.

Because of fresh seal generation, our evaluation is not quite deterministic. For instance, we have both $\langle \emptyset \rangle\, \nu x.x \Downarrow \langle \{k_1\} \rangle\, k_1$ and $\langle \emptyset \rangle\, \nu x.x \Downarrow \langle \{k_2\} \rangle\, k_2$ for $k_1 \neq k_2$. Nevertheless, we have the following property:

Property 2.1. Evaluation is deterministic modulo the names of freshly generated seals. That is, for any $\langle s \rangle\, e \Downarrow \langle t \rangle\, v$ and $\langle s \rangle\, e \Downarrow \langle t' \rangle\, v'$, we have $v = [\overline{k}/\overline{x}]e_0$ and $v' = [\overline{k'}/\overline{x}]e_0$ for some e_0 with $Seals(e_0) \subseteq s$, some $\overline{k}$ with $\{\overline{k}\} \subseteq t \setminus s$, and some $\overline{k'}$ with $\{\overline{k'}\} \subseteq t' \setminus s$.

PROOF. Straightforward induction on the derivation of $\langle s \rangle\, e \Downarrow \langle t \rangle\, v$. □

In what follows, we implicitly use the following properties of evaluation without explicitly referring to them.

Property 2.2. Every value evaluates only to itself. That is, for any s and v with $s \supseteq Seals(v)$, we have $\langle s \rangle\, v \Downarrow \langle s \rangle\, v$. Furthermore, if $\langle s \rangle\, v \Downarrow \langle t \rangle\, w$, then $t = s$ and $w = v$.

PROOF. Straightforward induction on the syntax of values. □

Property 2.3. Evaluation never decreases the seal set. That is, for any $\langle s \rangle\, e \Downarrow \langle t \rangle\, v$, we have $s \subseteq t$.

PROOF. Straightforward induction on the derivation of $\langle s \rangle\, e \Downarrow \langle t \rangle\, v$. □

3 Contextual Equivalence

In standard untyped λ-calculus, contextual equivalence for closed values[1] can be defined by saying that v and v' are contextually equivalent if $[v/x]e \Downarrow \iff [v'/x]e \Downarrow$ for any term e. In $\lambda_{\texttt{seal}}$, however, contextual equivalence cannot be defined for two values in isolation. For instance, consider $\lambda\{x\}_k.x + 1$ and $\lambda\{x\}_{k'}.x + 2$. Whether these values are equivalent or not depends on what values sealed under k or k' are known to the context. If the original terms which created k and k' were $\nu z.(\{2\}_z, \lambda\{x\}_z.x + 1)$ and $\nu z.(\{1\}_z, \lambda\{x\}_z.x + 2)$, for example, then the only values sealed under k or k' are 2 and 1, respectively. Thus, the equivalence above does hold. On the other hand, it does not hold if the terms were, say, $\nu z.(\{3\}_z, \lambda\{x\}_z.x + 1)$ and $\nu z.(\{4\}_z, \lambda\{x\}_z.x + 2)$. This observation that we have to consider multiple pairs of values at once leads to the following definition of contextual equivalence.

Definition 3.1. A *value relation* R is a set of pairs of values.

Definition 3.2. The *contextual equivalence* $\equiv$ is the largest relation among seal sets s and s' and value relations R, such that for any $(s, s', R) \in \equiv$ and for any $(\overline{v}, \overline{v'}) \in R$, we have the following properties.

1. $Seals(\overline{v}) \subseteq s$ and $Seals(\overline{v'}) \subseteq s'$.

[1] For the sake of simplicity, we focus on equivalence of closed values (as opposed to open expressions) in this paper. For open expressions e and e' with free variables $x_1, \ldots, x_n$, it suffices to consider the equivalence of $\lambda x_1. \ldots \lambda x_n.e$ and $\lambda x_1. \ldots \lambda x_n.e'$ instead.

$$\frac{Seals(e) \subseteq s}{\langle s\rangle\, \lambda x.e \Downarrow \langle s\rangle\, \lambda x.e}\text{(E-Lam)} \qquad \frac{\langle s\rangle\, e_1 \Downarrow \langle s_1\rangle\, \lambda x.e \quad \langle s_1\rangle\, e_2 \Downarrow \langle s_2\rangle\, v \quad \langle s_2\rangle\, [v/x]e \Downarrow \langle t\rangle\, w}{\langle s\rangle\, e_1\, e_2 \Downarrow \langle t\rangle\, w}\text{(E-App)}$$

$$\frac{}{\langle s\rangle\, c \Downarrow \langle s\rangle\, c}\text{(E-Const)} \qquad \frac{\langle s\rangle\, e_1 \Downarrow \langle s_1\rangle\, c_1 \quad \ldots \quad \langle s_{n-1}\rangle\, e_n \Downarrow \langle s_n\rangle\, c_n \quad [\![op(c_1,\ldots,c_n)]\!] = c}{\langle s\rangle\, op(e_1,\ldots,e_n) \Downarrow \langle s_n\rangle\, c}\text{(E-Prim)}$$

$$\frac{\langle s\rangle\, e_1 \Downarrow \langle s_1\rangle\, \texttt{true} \quad \langle s_1\rangle\, e_2 \Downarrow \langle t\rangle\, v}{\langle s\rangle\, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow \langle t\rangle\, v}\text{(E-Cond-True)} \qquad \frac{\langle s\rangle\, e_1 \Downarrow \langle s_1\rangle\, \texttt{false} \quad \langle s_1\rangle\, e_3 \Downarrow \langle t\rangle\, v}{\langle s\rangle\, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow \langle t\rangle\, v}\text{(E-Cond-False)}$$

$$\frac{\langle s\rangle\, e_1 \Downarrow \langle s_1\rangle\, v_1 \quad \ldots \quad \langle s_{n-1}\rangle\, e_n \Downarrow \langle s_n\rangle\, v_n}{\langle s\rangle\, (e_1,\ldots,e_n) \Downarrow \langle s_n\rangle\, (v_1,\ldots,v_n)}\text{(E-Tuple)} \qquad \frac{\langle s\rangle\, e \Downarrow \langle t\rangle\, (v_1,\ldots,v_n) \quad 1 \le i \le n}{\langle s\rangle\, \#_i(e) \Downarrow \langle t\rangle\, v_i}\text{(E-Proj)}$$

$$\frac{k \in s}{\langle s\rangle\, k \Downarrow \langle s\rangle\, k}\text{(E-Seal)} \qquad \frac{\langle s \uplus \{k\}\rangle\, [k/x]e \Downarrow \langle t\rangle\, v}{\langle s\rangle\, \nu x.e \Downarrow \langle t\rangle\, v}\text{(E-New)} \qquad \frac{\langle s\rangle\, e_1 \Downarrow \langle s_1\rangle\, v \quad \langle s_1\rangle\, e_2 \Downarrow \langle s_2\rangle\, k}{\langle s\rangle\, \{e_1\}_{e_2} \Downarrow \langle s_2\rangle\, \{v\}_k}\text{(E-Do-Seal)}$$

$$\frac{\langle s\rangle\, e_1 \Downarrow \langle s_1\rangle\, k \quad \langle s_1\rangle\, e_2 \Downarrow \langle s_2\rangle\, \{v\}_k \quad \langle s_2\rangle\, [v/x]e_3 \Downarrow \langle t\rangle\, w}{\langle s\rangle\, \texttt{let}\ \{x\}_{e_1} = e_2\ \texttt{in}\ e_3\ \texttt{else}\ e_4 \Downarrow \langle t\rangle\, w}\text{(E-Unseal-Succ)}$$

$$\frac{\langle s\rangle\, e_1 \Downarrow \langle s_1\rangle\, k_1 \quad \langle s_1\rangle\, e_2 \Downarrow \langle s_2\rangle\, \{v\}_{k_2} \quad k_1 \neq k_2 \quad \langle s_2\rangle\, e_4 \Downarrow \langle t\rangle\, w}{\langle s\rangle\, \texttt{let}\ \{x\}_{e_1} = e_2\ \texttt{in}\ e_3\ \texttt{else}\ e_4 \Downarrow \langle t\rangle\, w}\text{(E-Unseal-Fail)}$$

Figure 2. Semantics of $\lambda_{\texttt{seal}}$

2. $\langle s\rangle\, [\overline{v}/\overline{x}]e \Downarrow \iff \langle s'\rangle\, [\overline{v'}/\overline{x}]e \Downarrow$ for any e with $Seals(e) = \emptyset$.

We write $\langle s\rangle\, v_1,\ldots,v_n \equiv \langle s'\rangle\, v'_1,\ldots,v'_n$ for $(s,s',\{(v_1,v'_1),\ldots,(v_n,v'_n)\}) \in \equiv$. In order to lighten the notation, we do not enclose these $v_1,\ldots,v_n$ and $v'_1,\ldots,v'_n$ in parentheses. We also write $\langle s\rangle\, v \equiv_R \langle s'\rangle\, v'$ when $(v,v') \in R$ and $(s,s',R) \in \equiv$. Intuitively, it can be read as "value v with seal set s and value v' with seal set s' are contextually equivalent under contexts' knowledge R."

Note that no generality is lost by requiring $Seals(e) = \emptyset$ in the definition above: if e needs its own seals, it can freshly generate an arbitrary number of them by using ν; if e knows some of the seals $\overline{k}$ in s and $\overline{k'}$ in s', it suffices to require $(\overline{k},\overline{k'}) \in R$. Thus, our contextual equivalence subsumes standard contextual equivalence where a context knows none, all, or part of the seals (or, more generally, values involving the seals).

Example 3.3. Let $s = \{k\}$ and $s' = \{k'\}$. We have $\langle s\rangle\, \{2\}_k \equiv \langle s'\rangle\, \{1\}_{k'}$ since the context has no means to unseal the values sealed under k or k'. (A formal proof of this claim based on bisimulation will be given later.) We also have $\langle s\rangle\, \lambda\{x\}_k.x+1 \equiv \langle s'\rangle\, \lambda\{x\}_{k'}.x+2$ since the context cannot make up any values sealed under k or k'.

Furthermore, we have

$$\langle s\rangle\, \{2\}_k, \lambda\{x\}_k.x+1 \quad \equiv \quad \langle s'\rangle\, \{1\}_{k'}, \lambda\{x\}_{k'}.x+2$$

because applications of the functions to the sealed values yield the same integer 3. Similarly,

$$\langle s\rangle\, \{4\}_k, \lambda\{x\}_k.x+1 \quad \equiv \quad \langle s'\rangle\, \{5\}_{k'}, \lambda\{x\}_{k'}.x$$

holds. However,

$$\langle s\rangle\, \{2\}_k, \lambda\{x\}_k.x+1, \{4\}_k, \lambda\{x\}_k.x+1 \quad \not\equiv \quad \langle s'\rangle\, \{1\}_{k'}, \lambda\{x\}_{k'}.x+2, \{5\}_{k'}, \lambda\{x\}_{k'}.x$$

does not hold, because applications of the last functions to the first sealed values yield different integers 3 and 1.

As the last example shows, even if $(s,s',R_1) \in \equiv$ and $(s,s',R_2) \in \equiv$, we do not always have $(s,s',R_1 \cup R_2) \in \equiv$. Intuitively, this means that we should not confuse two worlds where the uses of seals are different. This is the reason why we defined $\equiv$ as a *set* of triples (s,s',R) rather than just a *function* that takes a pair (s,s') of seal sets and returns the "largest" set R of pairs of equivalent values.

Conversely, again as the examples above suggest, there are cases where $(s,s',R_1) \in \equiv$ and $(s,s',R_2) \in \equiv$ for $R_1 \subseteq R_2$. This implies that there is a partial order among the value relations R in contextual equivalence. We could alternatively define contextual equivalence only with such value relations that are maximal in this ordering, but this would just complicate the technicalities that follow.

We will use the following lemmas about contextual equivalence in what follows.

Lemma 3.4. Application, projection, fresh seal generation, and unsealing preserve contextual equivalence. That is:

1. For any $(u,u') \in R$ and $(v,v') \in R$ with $(s,s',R) \in \equiv$, if $\langle s\rangle\, uv \Downarrow \langle t\rangle\, w$ and $\langle s\rangle\, u'v' \Downarrow \langle t\rangle\, w'$, then $(t,t',R \cup \{(w,w')\}) \in \equiv$.

2. For any $((v_1,\ldots,v_n),(v'_1,\ldots,v'_n)) \in R$ with $(s,s',R) \in \equiv$, we have $(s,s',R \cup \{(v_i,v'_i)\}) \in \equiv$ for any $1 \le i \le n$.

3. For any $(s,s',R) \in \equiv$, we have $(s \uplus \{k\}, s' \uplus \{k'\}, R \uplus \{(k,k')\}) \in \equiv$ for any $k \notin s$ and $k' \notin s'$.

4. For any $(\{v\}_k, \{v'\}_{k'}) \in R$ and $(k,k') \in R$ with $(s,s',R) \in \equiv$, we have $(s,s',R \cup \{(v,v')\}) \in \equiv$.

PROOF. Take e as $\texttt{let}\ z = xy\ \texttt{in}\ e_0$, $\texttt{let}\ y = \#_i(x)\ \texttt{in}\ e_0$, $\nu x.e_0$, and $\texttt{let}\ \{z\}_x = y\ \texttt{in}\ e_0$, respectively, in the definition of $(s,s',R) \in \equiv$. □

Lemma 3.5. Contextually equivalent values put in the same value context yield contextually equivalent values. That is, for any $(s,s',R) \in \equiv$ and $(\overline{v},\overline{v'}) \in R$, and for any $w = [\overline{v}/\overline{x}]e_0$ and $w' = [\overline{v'}/\overline{x}]e_0$ with $Seals(e_0) = \emptyset$, we have $(s,s',R \cup \{(w,w')\}) \in \equiv$.

PROOF. Immediate from the definition of contextual equivalence, using the property of substitution that $[[\overline{v}/\overline{x}]e_0/x]e = [\overline{v}/\overline{x}]([e_0/x]e)$ when $\{\overline{x}\} \cap FV(e) = \emptyset$. □

Lemma 3.6. Any subset of contextually equivalent values are contextually equivalent. That is, for any $(s, s', R) \in \equiv$, we have $(s, s', S) \in \equiv$ for any $S \subseteq R$.

PROOF. Immediate from the definition of contextual equivalence. □

4 Bisimulation

Giving a direct proof of contextual equivalence is generally difficult, because the definition involves universal quantification over an infinite number of contexts. Thus, we want a more convenient tool for proving contextual equivalence. For this purpose, we define the notion of bisimulation as follows.

Definition 4.1. A *bisimulation* X is a relation among seal sets s and s' and value relations R such that every $(s, s', R) \in X$ satisfies the following conditions.

1. For each $(v, v') \in R$, we have $Seals(v) \subseteq s$ and $Seals(v') \subseteq s'$.
2. For each $(v, v') \in R$, v and v' are of the same kind. That is, both are functions, both are constants, both are tuples, both are seals, or both are sealed values.
3. For each $(c, c') \in R$, we have $c = c'$.
4. For each $((v_1, \ldots, v_n), (v'_1, \ldots, v'_{n'})) \in R$, we have $n = n'$ and $(s, s', R \cup \{(v_i, v'_i)\}) \in X$ for every $1 \leq i \leq n$.
5. For each $(k_1, k'_1) \in R$ and $(k_2, k'_2) \in R$, we have $k_1 = k_2 \iff k'_1 = k'_2$.
6. For each $(\{v\}_k, \{v'\}_{k'}) \in R$, we have either $(k, k') \in R$ and $(s, s', R \cup \{(v, v')\}) \in X$, or else $(k, k'') \notin R$ and $(k'', k') \notin R$ for every k''.
7. Take any $(\lambda x.e, \lambda x.e') \in R$. Take also any $\overline{k}$ and $\overline{k}'$ with $s \cap \{\overline{k}\} = s' \cap \{\overline{k}'\} = \emptyset$. Moreover, let $v = [\overline{u}/\overline{x}]d$ and $v' = [\overline{u}'/\overline{x}]d$ for any $(\overline{u}, \overline{u}') \in R \uplus \{(\overline{k}, \overline{k}')\}$ and $Seals(d) = \emptyset$. Then, we have $\langle s \uplus \{\overline{k}\}\rangle (\lambda x.e)v \Downarrow \iff \langle s' \uplus \{\overline{k}'\}\rangle (\lambda x.e')v' \Downarrow$. Furthermore, if $\langle s \uplus \{\overline{k}\}\rangle (\lambda x.e)v \Downarrow \langle t\rangle w$ and $\langle s' \uplus \{\overline{k}'\}\rangle (\lambda x.e')v' \Downarrow \langle t'\rangle w'$, then $(t, t', R \uplus \{(\overline{k}, \overline{k}')\} \cup \{(w, w')\}) \in X$.

For any bisimulation X, we write $\langle s\rangle v \, X_R \, \langle s'\rangle v'$ when $(v, v') \in R$ and $(s, s', R) \in X$. This can be read "values v and v' with seal sets s and s' are bisimilar under contexts' knowledge R."

The intuitions behind the definition of bisimulation are as follows. Each of the conditions excludes pairs of values that are distinguishable by a context (except for Condition 1, which just restricts the scoping of seals). Condition 2 excludes pairs of values of different kinds, e.g., 123 and $\lambda x.x$. Condition 3 excludes pairs of different constants. Condition 4 excludes pairs of tuples with distinguishable elements. Condition 5 excludes cases such as $(k, k') \in R$ and $(k, k'') \in R$ with $k' \neq k''$, for which contexts like $\texttt{let } \{x\}_y = \{()\}_z \texttt{ in } x \texttt{ else } \bot$ can distinguish the left-hand side (setting $y = z = k$) and the right-hand side (setting $y = k'$ and $z = k''$). Condition 6 excludes cases where (i) the context can unseal both of two sealed values whose contents are distinguishable, or (ii) the context can unseal only one of the two sealed values.

Condition 7, the most interesting one, is about what a context can do to distinguish two functions. Obviously, this will involve applying them to some arguments—but what arguments? Certainly not arbitrary terms, because in general a context has only a partial knowledge of (values involving) the seals in s and s'. All that a context can do for making up the arguments is to carry out some computation d using values $\overline{u}$ and $\overline{u}'$ from its knowledge. Therefore, the arguments have forms $[\overline{u}/\overline{x}]d$ and $[\overline{u}'/\overline{x}]d$.

An important and perhaps surprising point here is that it actually suffices to consider cases where these arguments are *values*. This restriction is useful and even crucial for proving bisimulation of functions: if the arguments $[\overline{u}/\overline{x}]d$ and $[\overline{u}'/\overline{x}]d$ were not values, we should evaluate them before applying the functions; in particular, if evaluation of one argument converges, then evaluation of the other argument must converge as well; proving this property amounts to proving the contextual equivalence of $\overline{u}$ and $\overline{u}'$, which was the whole purpose of our bisimulation!

Fortunately, our restriction of the arguments to values can be justified by the "fundamental property" proved in the next section, which says that the special forms $[\overline{u}/\overline{x}]d$ and $[\overline{u}'/\overline{x}]d$ are preserved by evaluation. The only change required as a result of this restriction is the addition of $\{(\overline{k}, \overline{k}')\}$ to knowledge R in Condition 7: it compensates for the fact that d can no longer be a fresh seal generation, while the context can still generate its own fresh seals $\overline{k}$ and $\overline{k}'$ when making up the arguments. Without such a change, our bisimulation would indeed be unsound: a counter-example would be $(\emptyset, \emptyset, \{(\lambda x.\{\texttt{true}\}_x, \lambda x.\{\texttt{false}\}_x)\})$, for which contexts like $\nu y.\texttt{let } \{z\}_y = [\,] y \texttt{ in if } z \texttt{ then } () \texttt{ else } \bot$ can distinguish the two functions.

The rest of Condition 7 is straightforward: the results w and w' of function application should also be bisimilar.

Example 4.2. Let $s = \{k\}$, $s' = \{k'\}$, and $R = \{(\{2\}_k, \{1\}_{k'})\}$. Then $\{(s, s', R)\}$ is a bisimulation, as can be seen by a straightforward check of the conditions above.

Example 4.3. Let $s = \{k_1, k_2\}$, $s' = \{k'\}$, and

$$\begin{array}{rcl} R & = & \{((\{2\}_{k_1}, \{4\}_{k'}), (\{1\}_{k_2}, \{5\}_{k'})), \\ & & \ (\{2\}_{k_1}, \{1\}_{k'}), \\ & & \ (\{4\}_{k_2}, \{5\}_{k'})\}. \end{array}$$

Then $\{(s, s', R)\}$ is a bisimulation. This example illustrates the fact that the number of seals may differ in the left-hand side and in the right-hand side of bisimulation. Note that the closure condition (Condition 4) in the definition of bisimulation demands that we include not only the original pairs, but also their corresponding components.

Example 4.4. Suppose we want to show that the pair $(\{2\}_k, \lambda\{x\}_k.x + 1)$ is bisimilar to $(\{1\}_{k'}, \lambda\{x\}_{k'}.x + 2))$, assuming that seals k and k' are not known to the context. Again, the closure conditions in the definition force us to include the corresponding components of the pairs (Condition 4), as well as the results of evaluating the second components applied to the first components (Condition 7); moreover, since Condition 7 allows the context to enrich the set of seals with arbitrary seals of its own, our bisimulation will consist of an infinite collection of similar sets, differing in the context's choice of seals.

Formally, let G be the following function on sets of pairs of seals:

$$\begin{array}{rcl} G\{(\overline{k}_0,\overline{k}'_0)\} & = & \{((\{2\}_k, \lambda\{x\}_k.x+1),\\ & & \quad (\{1\}_{k'}, \lambda\{x\}_{k'}.x+2)),\\ & & \quad (\{2\}_k, \{1\}_{k'}),\\ & & \quad (\lambda\{x\}_k.x+1, \lambda\{x\}_{k'}.x+2),\\ & & \quad (3,3)\}\\ & \cup & \{(\overline{k}_0,\overline{k}'_0)\} \end{array}$$

Then

$$X \quad = \quad \{(\{k,\overline{k}_0\},\{k',\overline{k}'_0\}, G\{(\overline{k}_0,\overline{k}'_0)\}) \mid k \notin \{\overline{k}_0\} \wedge k' \notin \{\overline{k}'_0\}\}$$

is a bisimulation. The only non-trivial work required to show this is checking Condition 7 for the pair $(\lambda\{x\}_k.x+1, \lambda\{x\}_{k'}.x+2) \in G\{(\overline{k}_0,\overline{k}'_0)\}$, for each $\overline{k}_0$ and $\overline{k}'_0$ with $k \notin \{\overline{k}_0\}$ and $k' \notin \{\overline{k}'_0\}$.

Consider any $v = [\overline{u}/\overline{x}]d$ and $v' = [\overline{u}'/\overline{x}]d$ with $(\overline{u},\overline{u}') \in G\{(\overline{k}_0,\overline{k}'_0)\} \uplus \{(\overline{k}_1,\overline{k}'_1)\}$ and $Seals(d) = \{k,\overline{k}_0\} \cap \{\overline{k}_1\} = \{k',\overline{k}'_0\} \cap \{\overline{k}'_1\} = \emptyset$. If the evaluations of $(\lambda\{x\}_k.x+1)v$ and $(\lambda\{x\}_{k'}.x+2)v'$ diverge, then the condition holds.

Let us focus on cases where the evaluation of $(\lambda\{x\}_k.x+1)v$ converges (without loss of generality, thanks to symmetry), that is, where v is of the form $\{w\}_k$. Then, either d is of the form $\{d_0\}_{x_i}$ and $u_i = k$, or else d is a variable x_i and $u_i = \{w\}_k$. However, the former case is impossible: k is not in the first projection of $G\{(\overline{k}_0,\overline{k}'_0)\}$ or $\{(\overline{k}_1,\overline{k}'_1)\}$ by their definitions. So we must be in the latter case.

Since the only element of the form $\{w\}_k$ in the first projection of $G\{(\overline{k}_0,\overline{k}'_0)\} \uplus \{(\overline{k}_1,\overline{k}'_1)\}$ is $\{2\}_k$ where the corresponding element in its second projection is $\{1\}_{k'}$, we have $v = \{2\}_k$ and $v' = \{1\}_{k'}$. Then, the only evaluations of $(\lambda\{x\}_k.x+1)v$ and $(\lambda\{x\}_{k'}.x+2)v'$ are

$$\langle \{k,\overline{k}_0\} \uplus \{\overline{k}_1\} \rangle\, (\lambda\{x\}_k.x+1)v \quad \Downarrow \quad \langle \{k,\overline{k}_0,\overline{k}_1\} \rangle\, 3$$

and

$$\langle \{k',\overline{k}'_0\} \uplus \{\overline{k}'_1\} \rangle\, (\lambda\{x\}_{k'}.x+2)v' \quad \Downarrow \quad \langle \{k',\overline{k}'_0,\overline{k}'_1\} \rangle\, 3.$$

Thus, the condition follows from

$$G\{(\overline{k}_0,\overline{k}'_0)\} \uplus \{(\overline{k}_1,\overline{k}'_1)\} \cup \{(3,3)\} \quad = \quad G\{(\overline{k}_0,\overline{k}'_0),(\overline{k}_1,\overline{k}'_1)\}$$

and

$$(\{k,\overline{k}_0,\overline{k}_1\},\{k',\overline{k}'_0,\overline{k}'_1\}, G\{(\overline{k}_0,\overline{k}'_0),(\overline{k}_1,\overline{k}'_1)\}) \in X.$$

Example 4.5 (Complex Numbers). Now let us show a bisimulation relating the two implementations of complex numbers in Section 1. First, let

$$\begin{array}{rcl} v & = & (\lambda(x,y).\{(x+0.0,y+0.0)\}_k,\\ & & \quad \lambda\{(x,y)\}_k.(x,y),\\ & & \quad \lambda(\{(x_1,y_1)\}_k,\{(x_2,y_2)\}_k).\\ & & \qquad \{(x_1\times x_2 - y_1\times y_2, x_1\times y_2 + y_1\times x_2)\}_k\\ v' & = & (\lambda(x,y).\{(\mathtt{sqrt}(x\times x + y\times y), \mathtt{atan2}(y,x))\}_{k'},\\ & & \quad \lambda\{(r,\theta)\}_{k'}.(r\times \mathtt{cos}\ \theta, r\times \mathtt{sin}\ \theta),\\ & & \quad \lambda(\{(r_1,\theta_1)\}_{k'},\{(r_2,\theta_2)\}_{k'}).\{(r_1\times r_2,\theta_1+\theta_2)\}_{k'}). \end{array}$$

The first component of each triple corresponds to the `from_re_and_im` functions in 1. The implementation in v just seals the x and y coordinates provided as arguments, after checking that they are indeed real numbers by attempting to add them to 0.0. The implementation in v' performs an appropriate change of representation before sealing. The second components correspond to the `to_re_and_im` functions in 1, and the third components to the `multiply` functions.

The construction of the bisimulation follows the same pattern as Example 4.4, except that the operator G is more interesting:

$$\begin{array}{rl} & G\{(\overline{k}_0,\overline{k}'_0)\} =\\ & \quad \{(v,v')\}\\ \cup & \quad \{(\lambda(x,y).\{(x+0.0,y+0.0)\}_k,\\ & \qquad \lambda(x,y).\{(\mathtt{sqrt}(x\times x+y\times y), \mathtt{atan2}(y,x))\}_{k'}),\\ & \qquad (\lambda\{(x,y)\}_k.(x,y),\\ & \qquad \lambda\{(r,\theta)\}_{k'}.(r\times \mathtt{cos}\ \theta, r\times \mathtt{sin}\ \theta)),\\ & \qquad (\lambda(\{(x_1,y_1)\}_k,\{(x_2,y_2)\}_k).\\ & \qquad\quad \{(x_1\times x_2 - y_1\times y_2, x_1\times y_2 + y_1\times x_2)\}_k,\\ & \qquad \lambda(\{(r_1,\theta_1)\}_{k'},\{(r_2,\theta_2)\}_{k'}).\{(r_1\times r_2,\theta_1+\theta_2)\}_{k'})\}\\ \cup & \quad \{((x,y),(x,y)) \mid x \text{ and } y \text{ are arbitrary real numbers}\}\\ \cup & \quad \{(\{(r\cos\theta, r\sin\theta)\}_k,\{(r,\theta)\}_{k'}) \mid r \geq 0\}\\ \cup & \quad \{(\overline{k}_0,\overline{k}'_0)\} \end{array}$$

Example 4.6 (Applicative vs. Generative Functors). In this example, we use bisimulation to show the equivalence of two instantiations of a generative functor, where generativity is modeled by fresh seal generation and the equivalence really depends on the generativity.

A functor is a parameterized module—a function from modules to modules. For example, a module implementing sets by binary trees can be parameterized by the type of elements and their comparison function. In the same imaginary ML-like language as in Section 1, such a functor might be written as follows:

```
functor Set(module Element : sig
              type t
              val less_than : t -> t -> bool
            end) =
  type elt = Element.t
  abstype set = Element.t tree
  let empty : set = Leaf
  let rec add : elt -> set -> set =
    fun x ->
      <check that x has type elt>;
      fun Leaf -> Node(x, Leaf, Leaf)
        | Node(y, l, r) ->
            if Element.less_than x y then
              Node(y, add x l, r)
            else if Element.less_than y x then
              Node(y, l, add x r)
            else Node(y, l, r)
  let rec is_elt_of : elt -> set -> bool =
    fun x ->
      fun Leaf -> false
        | Node(y, l, r) ->
            if Element.less_than x y then
              is_elt_of x l
            else if Element.less_than y x then
              is_elt_of x r
            else true
end
```

Now, consider the following three applications of this functor:

```
module IntSet1 =
  Set(module Element =
    type t = int
    let less_than : t -> t -> bool =
      fun x -> fun y -> (x <int y)
  end)
module IntSet2 =
```

```
  Set(module Element =
    type t = int
    let less_than : t -> t -> bool =
      fun x -> fun y -> (x <int y)
  end)
module IntSet3 =
  Set(module Element =
    type t = int
    let less_than : t -> t -> bool =
      fun x -> fun y -> (x >int y)
  end)
```

If the functor `Set` is applicative, the abstract type `IntSet3.set` becomes compatible with `IntSet1.set` and `IntSet2.set`, even though the comparison function of `IntSet3` is not compatible with that of `IntSet1` or `IntSet2`. As a result, (part of) their abstraction as sets of integers is lost: for instance, `IntSet2` and `IntSet3` are distinguished by a context like

$$C[\,] \;=\; \texttt{let s = [].add 7 ([].add 3 [].empty) in IntSet1.is\_elt\_of 7 s}$$

while they *should* be equivalent if considered just as two different implementations of integer sets.

This situation can be translated into $\lambda_{\texttt{seal}}$ as follows. First, the applicative functor can be implemented by the following function f, using a standard call-by-value fixed-point operator `fix` (which is definable since the language is untyped).

$$\begin{array}{l}
\lambda\texttt{lt}.\\
\quad(\{\texttt{nil}\}_k,\\
\quad\texttt{fix}(\lambda\texttt{add}.\,\lambda(x,\{y\}_k).\\
\qquad\texttt{if lt}(x,x)\ \texttt{then}\ \bot\ \texttt{else}\\
\qquad\texttt{if}\ y=\texttt{nil}\ \texttt{then}\ \{(x,\texttt{nil},\texttt{nil})\}_k\ \texttt{else}\\
\qquad\texttt{if lt}(x,\#_1(y))\ \texttt{then}\ \{(\#_1(y),\texttt{add}(x,\#_2(y)),\#_3(y))\}_k\ \texttt{else}\\
\qquad\texttt{if lt}(\#_1(y),x)\ \texttt{then}\ \{(\#_1(y),\#_2(y),\texttt{add}(x,\#_3(y)))\}_k\ \texttt{else}\\
\qquad\{y\}_k),\\
\quad\texttt{fix}(\lambda\texttt{is_elt_of}.\,\lambda(x,\{y\}_k).\\
\qquad\texttt{if}\ y=\texttt{nil}\ \texttt{then false else}\\
\qquad\texttt{if lt}(x,\#_1(y))\ \texttt{then is_elt_of}(x,\#_2(y))\ \texttt{else}\\
\qquad\texttt{if lt}(\#_1(y),x)\ \texttt{then is_elt_of}(x,\#_3(y))\ \texttt{else}\\
\qquad\texttt{true}))
\end{array}$$

Next, we translate the three applications of the functor into three applications of f to appropriate comparison functions:

$$\begin{array}{lcl}
\langle\{k\}\rangle\, f(\lambda(x,y).x <_{\texttt{int}} y) & \Downarrow & \langle\{k\}\rangle\, v_1\\
\langle\{k\}\rangle\, f(\lambda(x,y).x <_{\texttt{int}} y) & \Downarrow & \langle\{k\}\rangle\, v_2\\
\langle\{k\}\rangle\, f(\lambda(x,y).x >_{\texttt{int}} y) & \Downarrow & \langle\{k\}\rangle\, v_3
\end{array}$$

The values v_2 and v_3 are *not* contextually equivalent when the context knows v_1. That is, $(\{k\},\{k\},\{(v_1,v_1),(v_2,v_3)\}) \notin \equiv$. To see this, take $e = \#_3(x)\ (7, \#_2(y)\ (7, \#_2(y)\ (3, \#_1(y))))$, setting $x = v_1$, $y = v_2$ in the left hand side and $x = v_1, y = v_3$ in the right hand side.

Note that v_2 and v_3 *are* contextually equivalent if the context knows neither v_1, f, nor any other values involving the seal k. That is, $(\{k\},\{k\},\{(v_2,v_3)\}) \in \equiv$. Indeed, the context $C[\,]$ above uses `IntSet1` to distinguish `IntSet2` and `IntSet3`. Our definition of contextual equivalence as a set of relations (annotated with seal sets) gives a precise account for such subtle variations of contexts' knowledge.

On the other hand, if we take the `Set` functor to be generative, then `IntSet2` and `IntSet3` are contextually equivalent even if the context also knows `IntSet1`, since all the abstract types are incompatible with one another. This case can be modeled in $\lambda_{\texttt{seal}}$ by the following function g, which generates a fresh seal for each application instead of using the same seal k for all instantiations.

$$\begin{array}{l}
\lambda\texttt{lt}.\nu z.\\
\quad(\{\texttt{nil}\}_z,\\
\quad\texttt{fix}(\lambda\texttt{add}.\,\lambda(x,\{y\}_z).\ldots),\\
\quad\texttt{fix}(\lambda\texttt{is_elt_of}.\,\lambda(x,\{y\}_z).\ldots))
\end{array}$$

Consider the following three applications of g.

$$\begin{array}{lcl}
\langle\emptyset\rangle\, g(\lambda(x,y).x <_{\texttt{int}} y) & \Downarrow & \langle\{\{k_1\}\}\rangle\, w_1\\
\langle\{\{k_1\}\}\rangle\, g(\lambda(x,y).x <_{\texttt{int}} y) & \Downarrow & \langle\{\{k_1,k_2\}\}\rangle\, w_2\\
\langle\{\{k_1\}\}\rangle\, g(\lambda(x,y).x >_{\texttt{int}} y) & \Downarrow & \langle\{\{k_1,k_3\}\}\rangle\, w_3
\end{array}$$

Now w_2 and w_3 are bisimilar even if the context knows w_1. That is, there exists a bisimulation X such that $(\{\{k_1,k_2\},\{k_1,k_3\},R\,) \in X$ with $\{(w_1,w_1),(w_2,w_3)\} \subseteq R$. It is straightforward to construct this bisimulation in the same manner as Examples 4.4 and 4.5.

Example 4.7. Let us show that $\lambda x.\,(3,x)$ is bisimilar to itself. This example is technically tricker than previous ones, because arbitrary values provided by the context can appear verbatim within results. These results can again be passed as arguments and thus appear within yet larger results, etc. To achieve the required closure conditions, we need to reach a limit of this process. This can be accomplished by defining a bisimulation X inductively.

We require $(\emptyset,\emptyset,\emptyset) \in X$ as the (trivial) base case. The induction rule is as follows. Take any $(s,s',R\,) \in X$. Take any $\overline{w} = [\overline{v}/\overline{x}]\overline{e}$ and $\overline{w}' = [\overline{v}'/\overline{x}]\overline{e}$ with $(\overline{v},\overline{v}') \in R$ and $\mathit{Seals}(\overline{e}) = \emptyset$. Take any $t \supseteq s$ and $t' \supseteq s'$ of the forms $\{\overline{k}\}$ and $\{\overline{k}'\}$. Let

$$\begin{array}{rcl}
S & = & \{(\lambda x.\,(3,x),\lambda x.\,(3,x)),\\
& & \;((3,\overline{w}),(3,\overline{w}')),\\
& & \;(3,3),\\
& & \;(\overline{w},\overline{w}'),\\
& & \;(\overline{k},\overline{k}')\}.
\end{array}$$

We then require that $(t,t',T\,) \in X$ for any $T \subseteq S$. The bisimulation we want is the least X satisfying these conditions.

Intuitively, we have defined X so that the conditions of bisimulation—Condition 7, in particular—are immediately satisfied. The final technical twist $T \subseteq S$ is needed because the closure conditions in the definition of bisimulation add individual pairs of elements rather than adding their whole "deductive closures" at once.

Example 4.8 (Protocol Encoding). As a final illustration of the power of our bisimulation technique (and $\lambda_{\texttt{seal}}$ itself), let us consider a more challenging example. This example is an encoding of the protocol below, which is based on the key exchange protocol of Needham, Schroeder, and Lowe [14, 20].

$$\begin{array}{lllll}
1. & \mathrm{B} \to \mathrm{A} & : & B\\
2. & \mathrm{A} \to \mathrm{B} & : & \{N_A, A\}_{k_B}\\
3. & \mathrm{B} \to \mathrm{A} & : & \{N_A, N_B, B\}_{k_A}\\
4. & \mathrm{A} \to \mathrm{B} & : & \{N_B\}_{k_B}\\
5. & \mathrm{B} \to \mathrm{A} & : & \{i\}_{N_B}
\end{array}$$

In this protocol, A is a server accepting requests from good B and evil E. It is supposed to work as follows. (1) B sends its own name B to A. (2) A generates a fresh nonce N_A, pair it with its own name A, encrypts the pair with B's public key, and sends it to B. (3) B generates a fresh key N_B, tuples it with N_A and B, encrypts the tuple with A's public key, and sends it to A. (4) A encrypts N_B with B's public key and sends it to B. (5) B encrypts some secret integer i with N_B and sends it to A.

$$
\begin{aligned}
S \;=\;& \{(U,U'),(V,V'),(W,W'),\\
& \textit{— corresponding keys and constants known to the attacker}\\
& (\overline{k},\overline{k}'),(A,A),(B,B),(E,E),\\
& (\lambda x.\{x\}_{k_A},\lambda x.\{x\}_{k'_A}),(\lambda x.\{x\}_{k_B},\lambda x.\{x\}_{k'_B}),(k_E,k'_E),\\
& (\overline{w},\overline{w}'),(\{\overline{w}\}_{k_A},\{\overline{w}'\}_{k'_A}),(\{\overline{w}\}_{k_B},\{\overline{w}'\}_{k'_B}),\\
& \textit{— corresponding components from principal B at Step 1}\\
& (\lambda\{(x,y)\}_{k_B}.\mathtt{assert}(y=A);\nu z.(\{(x,z,B)\}_{k_A},\lambda\{z_0\}_{k_B}.\mathtt{assert}(z_0=z);\{i\}_z),\\
& \ \lambda\{(x,y)\}_{k'_B}.\mathtt{assert}(y=A);\nu z.(\{(x,z,B)\}_{k'_A},\lambda\{z_0\}_{k'_B}.\mathtt{assert}(z_0=z);\{j\}_z)),\\
& \textit{— corresponding components from principal A at Step 2, communicating with B}\\
& ((\{(\overline{k}_{AB},A)\}_{k_B},\lambda\{(y_0,z,x_0)\}_{k_A}.\mathtt{assert}(y_0=\overline{k}_{AB});\mathtt{assert}(x_0=B);\{z\}_{k_B}),\\
& \ (\{(\overline{k}'_{AB},A)\}_{k'_B},\lambda\{(y_0,z,x_0)\}_{k'_A}.\mathtt{assert}(y_0=\overline{k}'_{AB});\mathtt{assert}(x_0=B);\{z\}_{k'_B})),\\
& (\{(\overline{k}_{AB},A)\}_{k_B},\{(\overline{k}'_{AB},A)\}_{k'_B}),\\
& (\lambda\{(y_0,z,x_0)\}_{k_A}.\mathtt{assert}(y_0=\overline{k}_{AB});\mathtt{assert}(x_0=B);\{z\}_{k_B},\\
& \ \lambda\{(y_0,z,x_0)\}_{k'_A}.\mathtt{assert}(y_0=\overline{k}'_{AB});\mathtt{assert}(x_0=B);\{z\}_{k'_B}),\\
& \textit{— corresponding components from principal B at step 3, communicating with A}\\
& ((\{(\overline{k}_{AB},\overline{k}_B,B)\}_{k_A},\lambda\{z_0\}_{k_B}.\mathtt{assert}(z_0=\overline{k}_{AB});\{i\}_{\overline{k}_B}),\\
& \ (\{(\overline{k}'_{AB},\overline{k}'_B,B)\}_{k'_A},\lambda\{z_0\}_{k'_B}.\mathtt{assert}(z_0=\overline{k}'_{AB});\{j\}_{\overline{k}'_B})),\\
& (\{(\overline{k}_{AB},\overline{k}_B,B)\}_{k_A},\{(\overline{k}'_{AB},\overline{k}'_B,B)\}_{k'_A}),\\
& (\lambda\{z_0\}_{k_B}.\mathtt{assert}(z_0=\overline{k}_{AB});\{i\}_{\overline{k}_B},\\
& \ \lambda\{z_0\}_{k'_B}.\mathtt{assert}(z_0=\overline{k}'_{AB});\{j\}_{\overline{k}'_B}),\\
& \textit{— corresponding components from principal A at step 4, communicating with B}\\
& (\{\overline{k}_B\}_{k_B},\{\overline{k}'_B\}_{k'_B}),\\
& \textit{— corresponding components from principal B at step 5, communicating with A}\\
& (\{i\}_{\overline{k}_B},\{j\}_{\overline{k}'_B}),\\
& \textit{— corresponding components from principal A at Step 2, communicating with E}\\
& ((\{(\overline{k}_{AE},A)\}_{k_E},\lambda\{(y_0,z,x_0)\}_{k_A}.\mathtt{assert}(y_0=\overline{k}_{AE});\mathtt{assert}(x_0=E);\{z\}_{k_E}),\\
& \ (\{(\overline{k}'_{AE},A)\}_{k'_E},\lambda\{(y_0,z,x_0)\}_{k'_A}.\mathtt{assert}(y_0=\overline{k}'_{AE});\mathtt{assert}(x_0=E);\{z\}_{k'_E})),\\
& (\{(\overline{k}_{AE},A)\}_{k_E},\{(\overline{k}'_{AE},A)\}_{k'_E}),\\
& (\lambda\{(y_0,z,x_0)\}_{k_A}.\mathtt{assert}(y_0=\overline{k}_{AE});\mathtt{assert}(x_0=E);\{z\}_{k_E},\\
& \ \lambda\{(y_0,z,x_0)\}_{k'_A}.\mathtt{assert}(y_0=\overline{k}'_{AE});\mathtt{assert}(x_0=E);\{z\}_{k'_E}),\\
& ((\overline{k}_{AE},A),(\overline{k}'_{AE},A)),\\
& (\overline{k}_{AE},\overline{k}'_{AE}),\\
& \textit{— corresponding components from principal B at Step 3, communicating with E}\\
& ((\{(\overline{w},\overline{k}_B,B)\}_{k_A},\lambda\{z_0\}_{k_B}.\mathtt{assert}(z_0=\overline{k}_B);\{i\}_{\overline{k}_B}),\\
& \ (\{(\overline{w}',\overline{k}'_B,B)\}_{k'_A},\lambda\{z_0\}_{k'_B}.\mathtt{assert}(z_0=\overline{k}'_B);\{j\}_{\overline{k}'_B})),\\
& (\{(\overline{w},\overline{k}_B,B)\}_{k_A},\{(\overline{w}',\overline{k}'_B,B)\}_{k'_A}),\\
& (\lambda\{z_0\}_{k_B}.\mathtt{assert}(z_0=\overline{k}_B);\{i\}_{\overline{k}_B},\\
& \ \lambda\{z_0\}_{k'_B}.\mathtt{assert}(z_0=\overline{k}'_B);\{j\}_{\overline{k}'_B})),\\
& \textit{— corresponding components from principal A at Step 4, communicating with E}\\
& (\{\overline{w}\}_{k_E},\{\overline{w}'\}_{k'_E})\}
\end{aligned}
$$

Figure 3. Bisimulation for the Needham-Schroeder-Lowe protocol

The idea of the encoding is as follows. We use sealing, unsealing and fresh seal generation as (perfect) encryption, decryption, and fresh key generation. The whole system is expressed as a tuple of (functions representing) keys known to the attacker and terms U and V representing principals B and A.

$$W \quad = \quad (\lambda x.\{x\}_{k_A}, \lambda x.\{x\}_{k_B}, k_E, U, V)$$

Each principal is encoded as a pair of the last value it sent (if any) and a continuation function waiting to receive a next message. When the message is received, the function returns the next state of the principal. Communication occurs by a context applying these functions in an appropriate order (when the environment is behaving normally) or perhaps in some strange, arbitrary order (when the environment is under the control of a malicious attacker). Thus, contexts play the role of the network, scheduler, and attackers. More details about the encoding—including a more detailed justification of the claim that it *is* a reasonable encoding of the protocol above—can be found in previous work [30]. We write $\mathtt{assert}(e_1); e_2$ as syntactic sugar for $\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ \bot$.

$$\begin{array}{rcl}
U & = & (B, \lambda\{(x,y)\}_{k_B}.\mathtt{assert}(y = A); \\
& & \quad \nu z.(\{(x,z,B)\}_{k_A}, \\
& & \quad\quad \lambda\{z_0\}_{k_B}.\mathtt{assert}(z_0 = z); \\
& & \quad\quad\quad \{i\}_z)) \\
V & = & \lambda x.\mathtt{let}\ k_x = (\mathtt{if}\ x = B\ \mathtt{then}\ k_B\ \mathtt{else} \\
& & \quad\quad \mathtt{if}\ x = E\ \mathtt{then}\ k_E\ \mathtt{else}\ \bot)\ \mathtt{in} \\
& & \quad \nu y.(\{(y,A)\}_{k_x}, \\
& & \quad\quad \lambda\{(y_0,z,x_0)\}_{k_A}.\mathtt{assert}(y_0 = y); \\
& & \quad\quad\quad \mathtt{assert}(x_0 = x); \\
& & \quad\quad\quad \{z\}_{k_x}))
\end{array}$$

Now, take any integers i and j. We prove that the system W above (where the secret value sent from B to A is i) and the system W' below (where the secret is j) are bisimilar, which means that the protocol keeps i and j secret against attackers.

$$\begin{array}{rcl}
U' & = & (B, \lambda\{(x,y)\}_{k'_B}.\mathtt{assert}(y = A); \\
& & \quad \nu z.(\{(x,z,B)\}_{k'_A}, \\
& & \quad\quad \lambda\{z_0\}_{k'_B}.\mathtt{assert}(z_0 = z); \\
& & \quad\quad\quad \{j\}_z)) \\
V' & = & \lambda x.\mathtt{let}\ k_x = (\mathtt{if}\ x = B\ \mathtt{then}\ k'_B\ \mathtt{else} \\
& & \quad\quad \mathtt{if}\ x = E\ \mathtt{then}\ k'_E\ \mathtt{else}\ \bot)\ \mathtt{in} \\
& & \quad \nu y.(\{(y,A)\}_{k_x}, \\
& & \quad\quad \lambda\{(y_0,z,x_0)\}_{k'_A}.\mathtt{assert}(y_0 = y); \\
& & \quad\quad\quad \mathtt{assert}(x_0 = x); \\
& & \quad\quad\quad \{z\}_{k_x})) \\
W' & = & (\lambda x.\{x\}_{k'_A}, \lambda x.\{x\}_{k'_B}, k'_E, U', V')
\end{array}$$

The construction of the bisimulation X is by induction, following the same basic pattern as Example 4.7. The base case is $(\emptyset,\emptyset,\emptyset) \in X$. The induction rule is as follows. Take any $(s,s',R) \in X$. Take any $\overline{w} = [\overline{v}/\overline{x}]\overline{e}$ and $\overline{w}' = [\overline{v}'/\overline{x}]\overline{e}$ with $(\overline{v},\overline{v}') \in R$ and $\mathit{Seals}(\overline{e}) = \emptyset$. Take any $t \supseteq s$ and $t' \supseteq s'$ of the forms $\{k_A, k_B, k_E, \overline{k}_{AB}, \overline{k}_{AE}, \overline{k}_B, \overline{k}\}$ and $\{k'_A, k'_B, k'_E, \overline{k}'_{AB}, \overline{k}'_{AE}, \overline{k}'_B, \overline{k}'\}$. Then, $(t,t',T) \in X$ for any subset T of the set S given in Figure 3. It is routine to check the conditions of bisimulation for this X.

It is well known that the secrecy property does not hold for the original version of this protocol (i.e., without Lowe's fix), in which the third message is $\{N_A, N_B\}_{k_A}$ instead of $\{N_A, N_B, B\}_{k_A}$ (i.e., the B is missing). This flaw is mirrored in our setting as well: if we tried to construct a bisimulation for this version in the same way as above, it would fail to be a bisimulation for the following reason. Since we would have

$$(\{(\overline{w}, \overline{k}_B)\}_{k_A}, \{(\overline{w}', \overline{k}'_B)\}_{k'_A}) \in S$$

instead of

$$(\{(\overline{w}, \overline{k}_B, B)\}_{k_A}, \{(\overline{w}', \overline{k}'_B, B)\}_{k'_A}) \in S$$

along with $(\overline{k}_{AE}, \overline{k}'_{AE}) \in S$, we would have $(\{(\overline{k}_{AE}, \overline{k}_B)\}_{k_A}, \{(\overline{k}'_{AE}, \overline{k}'_B)\}_{k'_A}) \in S$ by taking $\overline{w} = \overline{k}_{AE}$ and $\overline{w}' = \overline{k}'_{AE}$ in the definition of X above. Since we would have

$$\begin{array}{c}
(\lambda\{(y_0,z)\}_{k_A}.\mathtt{assert}(y_0 = \overline{k}_{AE}); \{z\}_{k_E}, \\
\lambda\{(y_0,z)\}_{k'_A}.\mathtt{assert}(y_0 = \overline{k}'_{AE}); \{z\}_{k'_E}) \in S
\end{array}$$

as well instead of

$$\begin{array}{c}
(\lambda\{(y_0,z,x_0)\}_{k_A}.\mathtt{assert}(y_0 = \overline{k}_{AE}); \mathtt{assert}(x_0 = E); \{z\}_{k_E}, \\
\lambda\{(y_0,z,x_0)\}_{k'_A}.\mathtt{assert}(y_0 = \overline{k}'_{AE}); \mathtt{assert}(x_0 = E); \{z\}_{k'_E}) \in S
\end{array}$$

we should also have $(\{\overline{k}_B\}_{k_E}, \{\overline{k}'_B\}_{k'_E}) \in S$ by applying these functions to the previous ciphertexts, according the condition of bisimulation for functions (Condition 7). Furthermore, since $(k_E, k'_E) \in S$, we would need $(\overline{k}_B, \overline{k}'_B) \in S$ as well, according to the condition of bisimulation for sealed values (Condition 6). Then, since $(\{i\}_{\overline{k}_B}, \{j\}_{\overline{k}'_B}) \in S$, we should require $(i,j) \in S$. This contradicts with the condition of bisimulation for constants (Condition 3) if $i \neq j$. Observe how the same attack is prevented in the fixed version of this protocol: the assertion $\mathtt{assert}(x_0 = E)$ fails since x_0 is bound to B.

5 Soundness and Completeness

Bisimilarity, written $\sim$, is the largest bisimulation. It exists because the union of two bisimulations is always a bisimulation. We will need several simple lemmas about bisimulation in the development that follows.

Lemma 5.1 (Monotonicity). Take any bisimulation X. For any $(s,s',R) \in X$ and $(t,t',S) \in X$ with $R \subseteq S$, if $\langle s \rangle\, \nu\, X_R\, \langle s' \rangle\, \nu'$, then $\langle t \rangle\, \nu\, X_S\, \langle t' \rangle\, \nu'$.

PROOF. Immediate from the definitions of $\langle s \rangle\, \nu\, X_R\, \langle s' \rangle\, \nu'$ and $\langle t \rangle\, \nu\, X_S\, \langle t' \rangle\, \nu'$. □

Lemma 5.2 (Addition of Constants). Take any bisimulation X and $(s,s',R) \in X$. Then, $X \cup \{(s,s',R \cup \{(c,c)\})\}$ is a bisimulation for any constant c.

PROOF. Straightforward by checking the conditions of bisimulation. □

Lemma 5.3 (Addition of Fresh Seals). Take any bisimulation X and $(s,s',R) \in X$. Then, $X \cup \{(s \uplus \{k\}, s' \uplus \{k'\}, R \uplus \{(k,k')\})\}$ is a bisimulation for any $k \notin s$ and $k' \notin s'$.

PROOF. Straightforward by checking the conditions of bisimulation. □

We want to show that the bisimilarity $\sim$ coincides with the contextual equivalence $\equiv$. Since we defined $\sim$ by co-induction, the easy direction is showing that contextual equivalence implies bisimilarity.

Lemma 5.4 (Completeness of Bisimilarity). $\equiv \subseteq \sim$.

PROOF. Since $\sim$ is the greatest bisimulation, it suffices to check that $\equiv$ is a bisimulation, which is routine using the lemmas about contextual equivalence. □

Next, we need to prove soundness, i.e., that bisimilarity implies contextual equivalence. For this purpose, we define the following relation.

Definition 5.5 (Bisimilarity in Context). We define $\cong$ as

$$\{(s,s',R,[\overline{v}/\overline{x}]e_0,[\overline{v}'/\overline{x}]e_0) \mid \langle s\rangle\,\overline{v} \sim_R \langle s'\rangle\,\overline{v}' \wedge Seals(e_0)=\emptyset\}$$

where $\langle s\rangle\,\overline{v} \sim_R \langle s'\rangle\,\overline{v}'$ is a shorthand for $\langle s\rangle\,v_1 \sim_R \langle s'\rangle\,v_1' \wedge \ldots \wedge \langle s\rangle\,v_n \sim_R \langle s'\rangle\,v_n'$.

We write $\langle s\rangle\,e \cong_R \langle s'\rangle\,e'$ for $(s,s',R,e,e') \in \cong$. The intuition of this definition is: $\cong$ relates bisimilar values $\overline{v}$ and $\overline{v}'$ put in context e_0.

The two lemmas below are the key properties of our bisimulation. The first states that evaluation preserves $\cong$, the second that $\cong$ implies observational equivalence (i.e., if evaluation of one expression converges, then evaluation of the other expression also converges).

Lemma 5.6 (Fundamental Property, Part I). Suppose $\langle s_0\rangle\,e \cong_{R_0} \langle s_0'\rangle\,e'$. If $\langle s_0\rangle\,e \Downarrow \langle t\rangle\,w$ and $\langle s_0'\rangle\,e' \Downarrow \langle t'\rangle\,w'$, then $\langle t\rangle\,w \cong_R \langle t'\rangle\,w'$ for some $R \supseteq R_0$.

PROOF. See the full version [31]. □

Lemma 5.7 (Fundamental Property, Part II). If $\langle s_0\rangle\,e \cong_{R_0} \langle s_0'\rangle\,e'$, then $\langle s_0\rangle\,e \Downarrow \iff \langle s_0'\rangle\,e' \Downarrow$.

PROOF. See the full version [31]. □

An immediate consequence of the previous property is that bisimulation implies contextual equivalence.

Corollary 5.8 (Soundness of Bisimilarity). $\sim \subseteq \equiv$.

PROOF. By the definitions of $\equiv$ and $\cong$ with Lemma 5.7. □

Combining soundness and completeness, we obtain the main theorem about our bisimulation: that bisimilarity coincides with contextual equivalence.

Theorem 5.9. $\sim = \equiv$.

PROOF. By Lemma 5.4 and Corollary 5.8. □

6 Extension with Equality for Sealed Values

A number of variants of $\lambda_{\texttt{seal}}$ can be considered. For example, the version of $\lambda_{\texttt{seal}}$ in this paper does not allow a context to test two sealed values for equality. This is reasonable if the environment is a safe runtime system (where sealing can be implemented just by tagging) which disallows comparison of sealed values. It is unrealistic, however, to expect such a restriction in an arbitrary (perhaps hostile) environment, where sealing must be implemented by encryption. Fortunately, our technique is extends directly to such a modest change as adding equality for sealed values. For instance, it is straightforward to extend $\lambda_{\texttt{seal}}$ with syntactic equality $=_1$ for first-order values (including sealed values) along with an additional condition of bisimulation: $v_1 =_1 v_2 \iff v_1' =_1 v_2'$ for every $(v_1,v_1') \in R$ and $(v_2,v_2') \in R$. Then, it is also straightforward to prove the soundness and completeness of bisimilarity under this extension, with an additional lemma that $\cong$ respects $=_1$ (which can be proved by induction on the syntax of values being compared).

Of course, the more observations we allow, the more difficult it becomes to establish the equivalence of two given modules. For example, the two implementations of complex numbers given in the introduction are no longer equivalent (or bisimilar) under the extension above, because there are many polar representations of $0+0i$ while there is only one cartesian representation. So, for example, a context like

```
C[ ]  =  let x = [ ].from_re_and_im(0.0, 0.0) in
         let y = [ ].from_re_and_im(-1.0, 0.0) in
         x =1 [ ].multiply x y
```

would distinguish `CartesianComplex` and `PolarComplex`. To recover the equivalence, the polar representation of $0+0i$ must be standarized and checks inserted wherever it can be created:

```
let from_re_and_im =
  fun (x, y) ->
    let z =
      if x = 0.0 && y = 0.0 then (0.0, 0.0) else
      (sqrt(x * x + y * y), atan2(y, x)) in
    <seal z under k>
let multiply =
  fun (z1, z2) ->
    let (r1, t1) = <unseal z1 under k> in
    let (r2, t2) = <unseal z2 under k> in
    let z =
      if r1 = 0.0 || r2 = 0.0 then (0.0, 0.0) else
      (r1 * r2, t1 + t2) in
    <seal z under k>
```

7 Related Work

As discussed in the introduction, sealing was first proposed by Morris [18, 19] and has been revisited in more recent work on extending the "scope" (in both informal and technical senses) of type abstraction in various forms [7, 12, 26, 28].

Bisimulations have been studied extensively in process calculi. In particular, bisimulations for the spi-calculus [1, 2, 5, 6] are the most relevant to this work, because the perfect encryption in spi-calculus is very similar to dynamic sealing in our calculus. Our bisimulation is analogous to bisimulations for spi-calculus in that both keep track of the environment's knowledge. However, since processes and terms are different entities in spi-calculus, all the technicalities—i.e., definitions and proofs—must be developed separately for processes and terms. By contrast, our bisimulation is monolithic and more straightforward. Furthermore, it is possible even to encode and verify some (though not all) security protocols in our framework. The encoding naturally models the concurrency among principals and attackers (including so-called "necessarily parallel" attacks) by means of interleaving. Thanks to higher-order functions, we can also imitate public-key encryption without extending the calculus. See our previous work [30] for further discussion about this encoding of security protocols.

Another line of work on bisimulations in process calculi concerns techniques for lightening the burden of constructing a bisimulation—e.g., Milner and Sangiorgi's "bisimulation up to" [27]. It remains to be seen whether these techniques would be useful in our setting. Note that our operational semantics is built upon big-step evaluation (as opposed to small-step reduction) in the first

place, which cuts down the intermediate terms and reduces the size of a bisimulation.

Abramsky [4] studied *applicative bisimulation* for the λ-calculus. For functions $\lambda x.e$ and $\lambda x.e'$ to be bisimilar, it requires that $(\lambda x.e)d$ and $(\lambda x.e')d$ are observationally equivalent for any closed d, and that they evaluate to bisimilar values if the evaluations converge. Thus, it requires the two arguments to be the same, which actually makes the soundness proof harder [9]. We avoided this problem by allowing some difference between the arguments of functions in our bisimulation.

Jeffrey and Rathke [10] defined bisimulation for λ-calculus with name generation, of which our seal generation is an instance. Although their theory does distinguish private and public names, it lacks a proper mechanism to keep track of contexts' knowledge of name-involving values in general, such as functions containing names inside the bodies. As a result, they had to introduce additional language constructs—such as global references [10] or communication channels [11]—for the bisimulation to be sound. We solved this problem by using a set of relations (rather than a single relation) between values as a bisimulation, i.e., by considering multiple pairs of values at once.

A well-known method of proving the abstraction obtained by type abstraction is logical relations [15, 25]. Although they are traditionally defined on denotational models, they have recently been studied in the syntactic setting of term models as well [22, 23]. In previous work [30], we have defined syntactic logical relations for perfect encryption and used them to prove secrecy properties of security protocols. Although logical relations are analogous to bisimulations in that both relate corresponding values between two different programs, logical relations are defined by induction on types and cannot be applied in untyped settings. Moreover, logical relations in more sophisticated settings (such as recursive functions and recursive types) than simply typed λ-calculus tend to become rather complicated. Indeed, "keys encrypting keys" (as in security protocols) required non-trivial extension in the logical relations above, while they imposed no difficulty to our bisimulation in this paper.

8 Conclusions

We have defined a bisimulation for $\lambda_{\texttt{seal}}$ and proved its soundness and completeness with respect to contextual equivalence.

There are several directions for future work. One is to apply our bisimulation to more examples, e.g., to prove the full abstraction of our translation of type abstraction into dynamic sealing—indeed, this was actually the original motivation for the present work. When the target language is untyped, the translation of source term $\vdash M : \tau$ can be given as $\texttt{let}\ x = \mathit{erase}(M)\ \texttt{in}\ E^+_\emptyset(x,\tau)$, where E^+ is defined like Figure 4 along with its dual E^- in a type-directed manner. Intuitively, E^+ is a "firewall" that protects terms from contexts, where E^- is a "sandbox" that protects contexts from terms. Bisimulation would help proving properties of this translation. We may also be able to use such an interpretation of type abstraction by dynamic sealing as a (both formal and informal) basis for reasoning about type abstraction in broader settings.

Another possibility is to define and use bisimulation for other forms of information hiding, such as type abstraction. Our treatment of seals are analogous to the treatment of generative names in general [24, 29], of which abstract types are an instance as soon as they escape from their scope (by communication [28], for example). Thus, it would be possible to define bisimulation for type abstraction in a similar manner to the definition of our bisimulation for dynamic sealing. This is interesting because such bisimulation may be complete with respect to contextual equivalence as in this work, while it is difficult to obtain complete logical relations for type abstraction [22, 23].

$$
\begin{array}{lcl}
E^+_\rho(x,\texttt{bool}) & = & x \\
E^+_\rho(x,\tau_1\times\ldots\times\tau_n) & = & \texttt{let}\ (y_1,\ldots,y_n) = x\ \texttt{in} \\
 & & (E^+_\rho(y_1,\tau_1),\ldots,E^+_\rho(y_n,\tau_n)) \\
E^+_\rho(x,\tau\to\sigma) & = & \lambda y.\,\texttt{let}\ z = xE^-_\rho(y,\tau)\ \texttt{in}\ E^+_\rho(z,\sigma) \\
E^+_\rho(x,\forall\alpha.\,\tau) & = & \lambda y.\,\texttt{let}\ z = x()\ \texttt{in}\ E^+_\rho(z,\tau) \\
E^+_\rho(x,\exists\alpha.\,\tau) & = & \nu z.\,E^+_{\rho,\alpha\mapsto z}(x,\tau) \\
E^+_\rho(x,\alpha) & = & \{x\}_{\rho(\alpha)} \\
E^+_\rho(x,\alpha) & = & x \qquad\qquad \text{if } \alpha\notin \mathit{Dom}(\rho) \\
 & & \\
E^-_\rho(x,\texttt{bool}) & = & \texttt{if}\ x\ \texttt{then true else false} \\
E^-_\rho(x,\tau_1\times\ldots\times\tau_n) & = & \texttt{let}\ (y_1,\ldots,y_n) = x\ \texttt{in} \\
 & & (E^-_\rho(y_1,\tau_1),\ldots,E^-_\rho(y_n,\tau_n)) \\
E^-_\rho(x,\tau\to\sigma) & = & \lambda y.\,\texttt{let}\ z = xE^+_\rho(y,\tau)\ \texttt{in}\ E^-_\rho(z,\sigma) \\
E^-_\rho(x,\forall\alpha.\,\tau) & = & \lambda y.\,\nu z.\,E^-_{\rho,\alpha\mapsto z}(x,\tau) \\
E^-_\rho(x,\exists\alpha.\,\tau) & = & E^-_\rho(x,\tau) \\
E^-_\rho(x,\alpha) & = & \texttt{let}\ \{y\}_{\rho(\alpha)} = x\ \texttt{in}\ y\ \texttt{else}\ \bot \\
E^-_\rho(x,\alpha) & = & x \qquad\qquad \text{if } \alpha\notin \mathit{Dom}(\rho)
\end{array}
$$

Figure 4. Translation of type abstraction into dynamic sealing

Mechanical support for bisimulation proofs is also of natural interest. Full automation is hopeless, since general cases subsume the halting problem (i.e., whether the evaluation of a λ-expression converges or diverges), but many of the conditions of bisimulation are easy to check or satisfy by adding elements to the bisimulation. One challenging point would be the case analysis on function arguments $[\overline{u}/\overline{x}]d$ and $[\overline{u}'/\overline{x}]d$ in Condition 7, shown in detail in Example 4.4.

Acknowledgements

We would like to thank Martín Abadi, Andre Scedrov, Naoki Kobayashi, and the members of Programming Language Club at the University of Pennsylvania for suggestions and support throughout the development of this work.

9 References

[1] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 104–115, 2001.

[2] Martín Abadi and Andrew D. Gordon. A bisimulation method for cryptographic protocols. *Nordic Journal of Computing*, 5:267–303, 1998. Preliminary version appeared in *Programming Languages and Systems – ESOP'98, 7th European Symposium on Programming, Lecture Notes in Computer Science*, Springer-Verlag, vol. 1381, pages 12–26, 1998.

[3] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999. Preliminary version appeared in *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pp. 36–47, 1997.

[4] Samson Abramsky. The lazy lambda calculus. In David A. Turner, editor, *Research Topics in Functional Programming*, pages 65–117. Addison Wesley, 1990.

[5] Michele Boreale, Rocco De Nicola, and Rosario Pugliese. Proof techniques for cryptographic processes. *SIAM Journal on Computing*, 31(3):947–986, 2002. Preliminary version appeared in *14th Annual IEEE Symposium on Logic in Computer Science*, pp. 157–166, 1999.

[6] Johannes Borgström and Uwe Nestmann. On bisimulations for the spi calculus. In *9th International Conference on Algebraic Methodology and Software Technology*, volume 2422 of *Lecture Notes in Computer Science*, pages 287–303. Springer-Verlag, 2002.

[7] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 236–249, 2003.

[8] Dan Grossman, Greg Morrisett, and Steve Zdancewic. Syntactic type abstraction. *ACM Transactions on Programming Languages and Systems*, 22(6):1037–1080, 2000.

[9] Douglas J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2), 1996.

[10] Alan Jeffrey and Julian Rathke. Towards a theory of bisimulation for local names. In *14th Annual IEEE Symposium on Logic in Computer Science*, pages 56–66, 1999.

[11] Alan Jeffrey and Julian Rathke. A theory of bisimulation for a fragment of concurrent ML with local names. *Theoretical Computer Science*, 2003. To appear. An extended abstract appeared in *15th Annual IEEE Symposium on Logic in Computer Science*, pp. 311–321, 2000.

[12] James J. Leifer, Gilles Peskine, Peter Sewell, and Keith Wansbrough. Global abstraction-safe marshalling with hash types. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 87–98, 2003.

[13] Barbara Liskov. A history of CLU. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, pages 133–147, 1993.

[14] Gavin Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, 1995.

[15] John C. Mitchell. On the equivalence of data representations. In *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 305–330. Academic Press, 1991.

[16] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential types. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988. Preliminary version appeared in *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 37–51, 1985.

[17] James H. Morris Jr. *Lambda-Calculus Models of Programming Languages*. PhD thesis, Massachusetts Institute of Technology, 1968.

[18] James H. Morris Jr. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, 1973.

[19] James H. Morris Jr. Types are not sets. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 120–124, 1973.

[20] Roger Needham and Michael Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[21] Benjamin Pierce and Eijiro Sumii. Relating cryptography and polymorphism, 2000. Manuscript. `http://www.yl.is.s.u-tokyo.ac.jp/~sumii/pub/`.

[22] Andrew M. Pitts. Existential types: Logical relations and operational equivalence. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 309–326. Springer-Verlag, 1998.

[23] Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000. Preliminary version appeared in *HOOTS II Second Workshop on Higher-Order Operational Techniques in Semantics*, *Electronic Notes in Theoretical Computer Science*, vol. 10, 1998.

[24] Andrew M. Pitts and Ian Stark. Observable properties of higher order functions that dynamically create local names, or: what's new? In *Mathematical Foundations of Computer Science*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, 1993.

[25] John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congres*, pages 513–523, 1983.

[26] Andreas Rossberg. Generativity and dynamic opacity for abstract types. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 241–252, 2003.

[27] Davide Sangiorgi and Robin Milner. The problem of "weak bisimulation up to". In *CONCUR '92, Third International Conference on Concurrency Theory*, volume 630 of *Lecture Notes in Computer Science*, pages 32–46. Springer-Verlag, 1992.

[28] Peter Sewell. Modules, abstract types, and distributed versioning. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 236–247, 2001.

[29] Ian Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, 1994. `http://www.dcs.ed.ac.uk/home/stark/publications/thesis.html`.

[30] Eijiro Sumii and Benjamin C. Pierce. Logical relations for encryption. In *14th IEEE Computer Security Foundations Workshop*, pages 256–269, 2001. Long version to appear in the Journal of Computer Security.

[31] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for dynamic sealing. `http://www.cis.upenn.edu/~sumii/`, 2003.

[32] Steve Zdancewic, Dan Grossman, and Greg Morrisett. Principals in programming languages: A syntactic proof technique. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, pages 197–207, 1999.